

Building User Interfaces

React 2

Building w/ React

Professor Yuhang Zhao

Setting up a React project

In the development environment

What you will need: *terminal, coding environment, Node.js*

```
npm install -g create-react-app
```

```
npx create-react-app <your-app-name>
```

```
cd <your-app-name>
```

```
npm start
```

Setting up a React project

In a sandbox

Simple React application:¹⁸

1. set sandbox settings to Babel preprocessor
2. import react and react-dom CDNs

React project:

— Create a project using a template / upload your project

¹⁸[See example in CodePen](#)

What we will learn today?

- Using Component Libraries with React
- Component development and reuse
- Dataflow among components

Using Component Libraries with React

Refresher: What are Component Libraries?¹

Definition: Software libraries that abstract away the low-level CSS implementation of user-facing elements.

Some popular libraries:

- * Bootstrap
- * Foundation
- * Semantic UI
- * Pure
- * UIKit

¹react-bootstrap

Integrating Bootstrap into React

Three methods:

1. Using the CDN
2. Bootstrap dependency
3. React Bootstrap package 🙋 *preferred method*

CDN-based Use

As we did to use it with JS, add to public/index.html:

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-gg0yR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">

...

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"
integrity="sha384-U02eT0CpHqdSJQ6hJty5KVphtPhzWj9W01c1HTMGA3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script>
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy60rQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
```


Bootstrap Dependency

Install Bootstrap as a dependency:

```
npm install bootstrap
```

Include in your app's entry JS file, e.g., `src/index.js`:

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

React Bootstrap packages

Using: react-bootstrap¹:

```
npm install react-bootstrap bootstrap
```

```
import { Button } from 'react-bootstrap';
```

Using reactstrap²:

```
npm install reactstrap react react-dom
```

```
import { Button } from 'reactstrap';
```

¹react-bootstrap

²reactstrap

A Very Simple React App³

```
ReactDOM.render(  
  <h1 className="jumbotron">Welcome to my Page!</h1>,  
  document.getElementById('welcome') );
```

³See in CodePen

Back to My Home Page Example⁴

```
<Jumbotron>
  <h1>Welcome to my page!</h1>
  <p>You can download my resume below!</p>
  <Button variant="info" size="large" onClick={this.open}>Download</Button>
  <Modal show={this.state.showModal} onHide={this.close}>
    <Modal.Body><h3>Downloading...</h3></Modal.Body>
    <Modal.Footer><Button onClick={this.close}>Close</Button></Modal.Footer>
  </Modal>
</Jumbotron>
```

⁴ [See in CodePen](#)

Component Development and Reuse

Refresher: React.Component

Definition: A React component is a function or class that accepts an input and returns a React element.

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

Refresher: React.Component, Continued

Components work like JS functions; they accept props and return React elements that correspond to what will be rendered in the DOM.

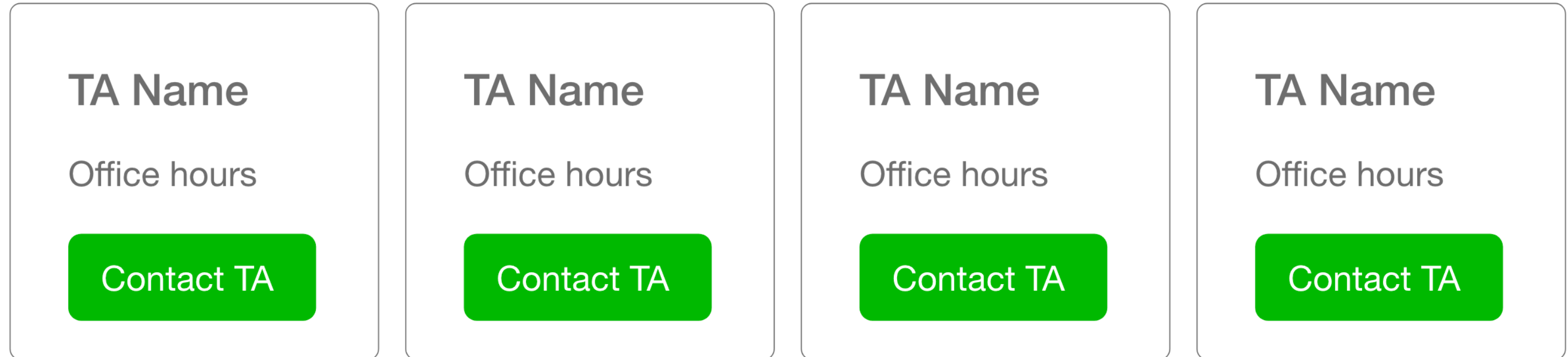
Each component is encapsulated (one component per file) and can operate independently, affording modularity.

"Thinking in React"⁵

1. Mock-up design
2. Break the UI into a component hierarchy
3. Build a static version
4. Identify the minimal set of mutable state
5. Identify where your state should live
6. Add inverse data flow

⁵ ReactJS.org: [Thinking in React](#)

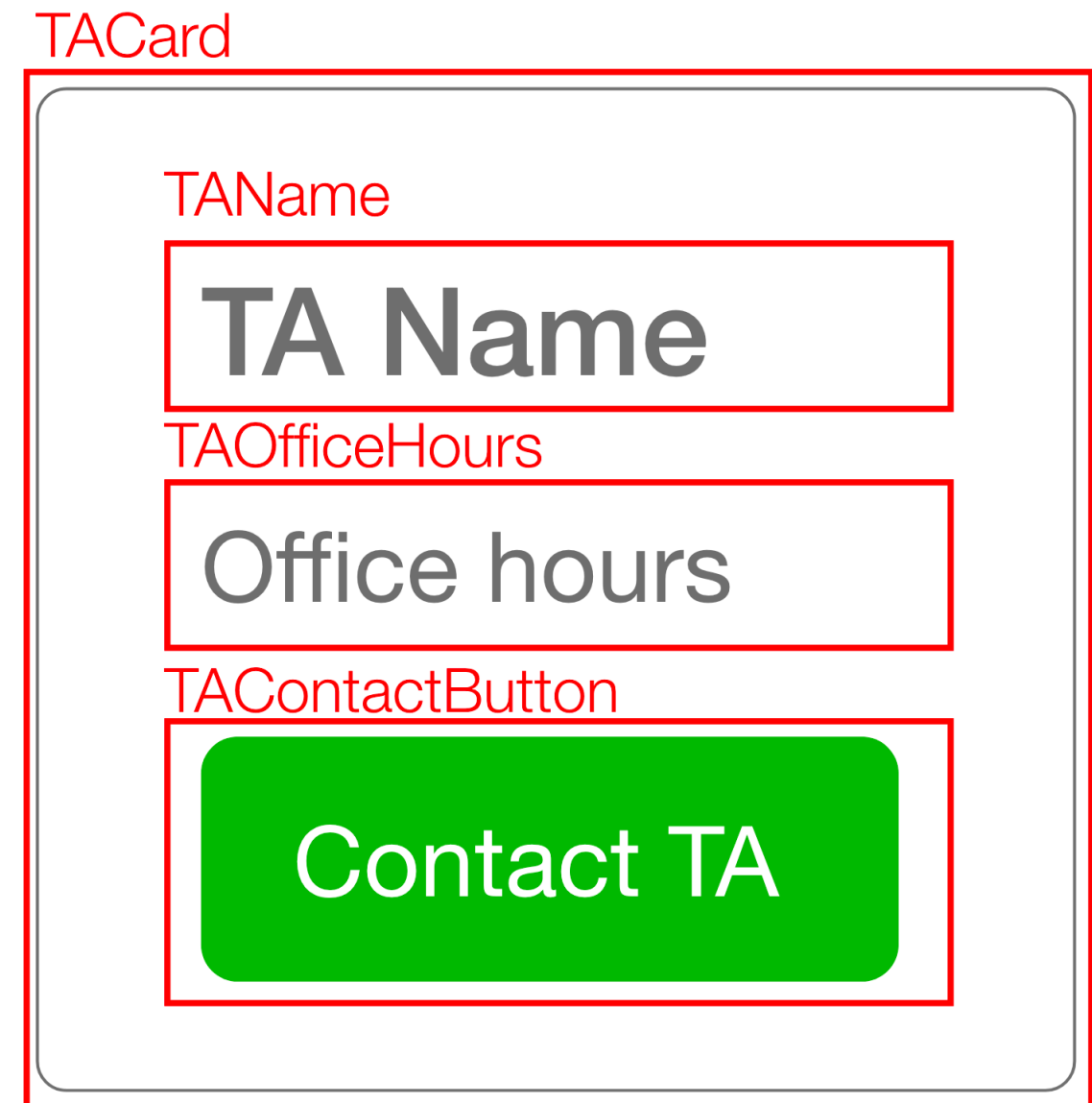
Step 1: Mock-up design



Step 2: Break UI into Component Hierarchy

- TACard
 - TAName
 - TAOfficeHours
 - TAContactButton

Note: This example is only illustrative. In a real development scenario, we would not dedicate components to, e.g., single text fields.



In-class activity

Write out or draw the component hierarchy for the elements shown in the screenshot above and listed below (submit to Quiz 4):

- NASA_logo
- user_name
- user_ID
- tweet_content
- tweet_time
- button_respond
- button_retweet
- button_like



NASA @NASA

Calling all students! Help design a 3-D space container for astronauts. More: go.nasa.gov/1HfitjM @K12FutureE
amp.twimg.com/v/e751b9a8-cb1...

2h ago



Step 3: Build A Static Version⁶

```
<div className={"col-sm-3"}>
  <div className={"card border-default mx-1"}>
    <div className="card-body">
      <h6 className="TA-name">Andy Schoen</h6>
      <p className="TA-office-hours">Tue, 3-5 pm</p>
      <button className={"btn btn-success"}>{"Contact Andy"}</button>
    </div>
  </div>
</div>
```

⁶[See in StackBlitz](#) ↩

Step 4: Identify the Minimal Mutable State

- TACard — Card container
 - TName — TA name text
 - TAOfficeHours — TA office hours text
 - TAContactButton — TA button text

We don't actually need to use state for any of these!

Step 4: Identify the Minimal Mutable State

Let's say that the button will change colors if the user contacts a TA to keep track of who has been contacted.

- TACard
 - TAName
 - TAOfficeHours
 - TAContactButton
 - TAContactButtonPressed — `this.state {...}`

Step 5: Identify Where the State Should Live

There are three options:

1. App keeps track of the button state for each `TAContactButton`
2. `TACard` keeps track of the button state for its `TAContactButton`
3. A dedicated `TAContactButton` component tracks its own state

All options will work, #3 is most aligned with *the React way*.

Step 6: Add Inverse Data Flow (more on this in a bit)

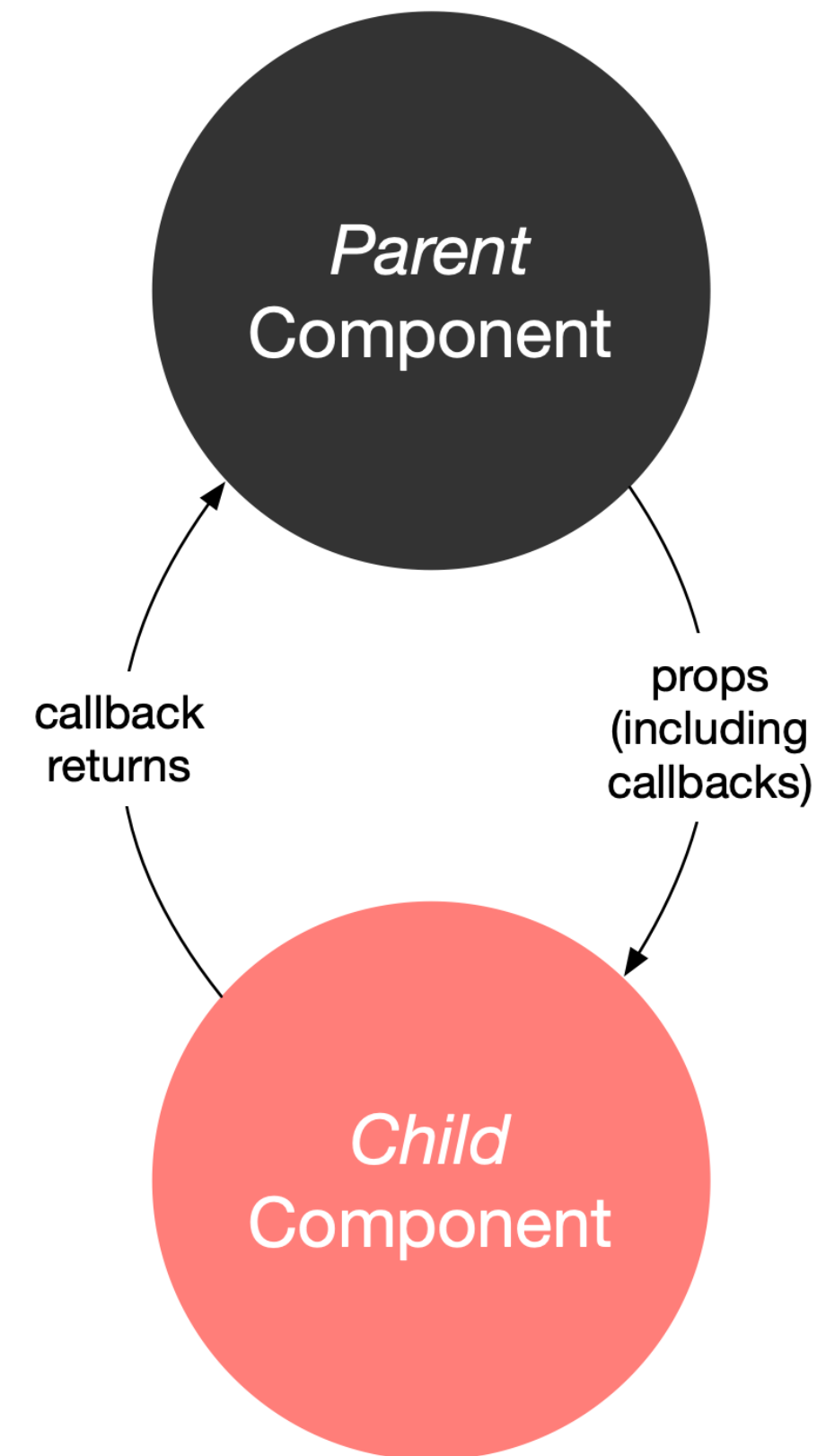
- App
 - CardComponent
 - ButtonComponent
 - CardComponent
 - ButtonComponent

Information Flow in React ⁷

Information flow is achieved using props and callbacks:

1. Parents pass props (including *callbacks*) to children
2. When executed, e.g., by being triggered by events, callbacks return information to the parent

More on this later in the lecture...



⁷[See in StackBlitz](#) ↩

Parent (CardComponent) callback:

```
<ButtonComponent  
  TAContectButtonText={this.state.TAContectButtonText}  
  callbackFromCard={this.buttonCallback}/>
```

```
buttonCallback = (dataFromButton) => {  
  // This acts on the callback from the button  
  dataFromButton  
    ? this.setState({ buttonCallBackMessage: '' })  
    : this.setState({ buttonCallBackMessage: 'Connecting...' });  
};
```

Child (ButtonComponent) response (called by onClick):

```
this.props.callbackFromCard(this.state.TAContectButtonPressed);
```

More on Components

- Class vs. functional components
- Fragments
- Controlled vs. uncontrolled components
- Passing methods through props

Class vs. Functional Components

A React Class component:

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

A React functional component (essentially a JS function):

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

Benefits of Functional Components

- *Easier to write, read, and debug*
- *More efficient to write and to run, as functional components involve less code*

Pro Tip 1: Before React 16.8, functional components are stateless. But it is NOT true any more! Now you can use State via *Hooks*!

Pro Tip 2: Create a components folder for Class and functional components, including one Component per file, to maximize modularity and code reuse.

Fragments⁸

Definition: Fragments are React constructs that can group child components without adding extra nodes to the DOM.

Benefits: Shortens code, reduces DOM complexity, and increases efficiency.

⁸ ReactJS.org: [Fragments](#)

An example

Imagine creating a table ...

```
class Table extends React.Component {  
  render() {  
    return (  
      <table>  
        <tr>  
          <Columns />  
        </tr>  
      </table>  
    );  
  }  
}
```


... and using a Columns component...

```
class Columns extends React.Component {  
  render() {  
    return (  
      <div>  
        <td>Hello</td>  
        <td>World</td>  
      </div>  
    );  
  }  
}
```

... the output of `<Table />` would be:

```
<table>
  <tr>
    <div>
      <td>Hello</td>
      <td>World</td>
    </div>
  </tr>
</table>
```

Do you see a problem here?

<div>s explosion! Fragments address this problem using the construct `<React.Fragment>` or `<>`.⁹

```
class Columns extends React.Component {  
  render() {  
    return (  
      <React.Fragment>  
        <td>Hello</td>  
        <td>World</td>  
      </React.Fragment>  
    );  
  }  
}
```

⁹[See example in CodePen](#)

... this React code would produce the following DOM representation:

```
<table>  
  <tr>  
    <td>Hello</td>  
    <td>World</td>  
  </tr>  
</table>
```

Look, no undesired divs!

Controlled vs. Uncontrolled Components ¹³

Definition: The states of *controlled* components are managed by React. User input elements, e.g., `<input>`, `<textarea>`, `<select>`, are *uncontrolled* when they manage their own states. We can use refs to give React access to DOM elements.

```
constructor(props) {  
  super(props);  
  this.input = React.createRef();  
}
```

```
<input type="text" ref={this.input} />
```

¹³[See on CodePen](#)

User input elements can also be handled in a *controlled* way, where the React state is the "single source of truth."¹⁴

```
constructor(props) {  
  super(props);  
  this.state = {value: ''};  
  this.handleChange = this.handleChange.bind(this);  
}  
handleChange(event) {  
  this.setState({value: event.target.value});  
}  
  
<input type="text" value={this.state.value} onChange={this.handleChange} />
```

¹⁴[See on codepen](#)

Passing Functions Through Props

We have been using props to pass data into child components, but props can also be used to pass *functions*, *event handlers*, and *event callbacks* (more on this in a little bit).

```
<button onClick={this.handleClick}>
```

Another Example ¹¹

In the constructor() of App:

```
this.state = {buttonColor: 'btn btn-success' };  
this.updateColor = this.updateColor.bind(this);
```

In the render():

```
<ContactButton buttonColor={this.state.buttonColor}  
updateButtonColor={this.updateColor}/>
```

An independent function:

```
updateColor() {  
  this.setState({  
    buttonColor : 'btn btn-danger'  
  });  
}
```

¹¹ See on StackBlitz

In the ContactButton class:

```
<React.Fragment>  
  <button class={props.buttonColor} onClick={props.updateButtonColor}>  
    Contact Customer Service  
  </button>  
</React.Fragment>
```

App gives ContactButton access to its updateColor() function by passing it through props.

```
this.bind()
```

Definition: *Binding*, through `this.<functionName>.bind(this)`, clarifies that the scope of the function that is passed to children component is within the parent component.

```
this.updateColor = this.updateColor.bind(this);
```

...

```
<ContactButton buttonColor={this.state.buttonColor}  
updateButtonColor={this.updateColor}/>
```

Pro Tip 1: Binding is usually done in the constructor(), but it can also be done within render(), although render() creates a new function every time the component renders (inefficient).

Pro Tip 2: Make sure that you are not *calling* the function, which will call the function every time components are rendered, but instead *passing* the function.

Example of *calling* (top) and *passing* (bottom) functions:

```
<button onClick={this.updateColor()}>Contact Customer Service</button>
```

```
<button onClick={this.updateColor}>Contact Customer Service</button>
```

Dataflow between Components

Why is dataflow necessary?¹²

Interactivity, modularity, and hierarchical construction necessitate dataflow among components.

- App
 - CardComponent
 - CardComponent

Can be *parent-to-child*, *child-to-parent*, and *sibling-to-sibling*.

¹² Ruth Pardee: Passing Data between React Components

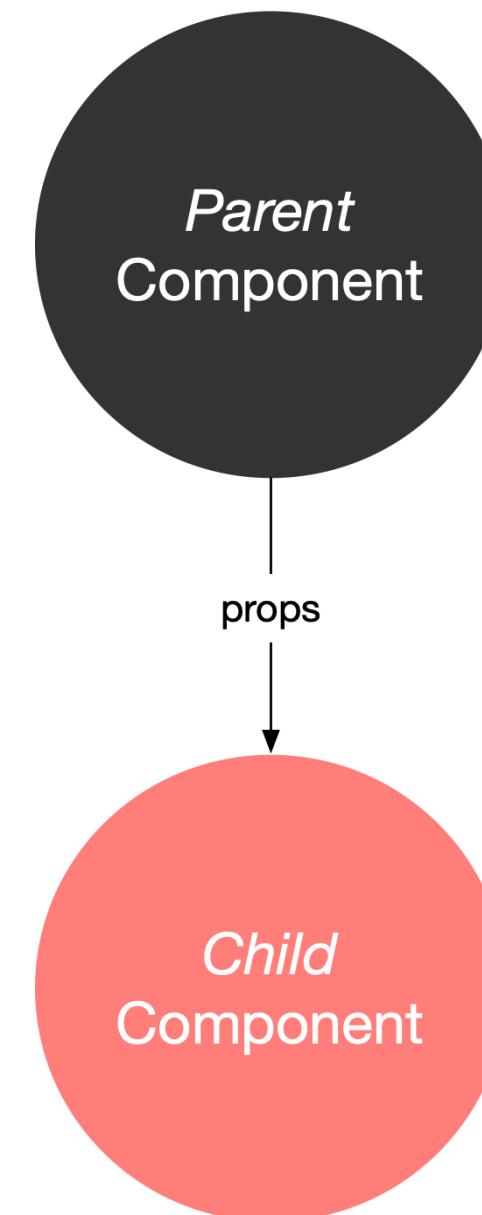
Parent-to-Child Dataflow

This is the easiest case — use props!

```
<CardComponent TAContactButtonText={"Contact John"}/>
```

```
constructor(props) {  
  super(props);  
  this.state = {  
    TAContactButtonText:  
    this.props.TAContactButtonText,  
  };  
}
```

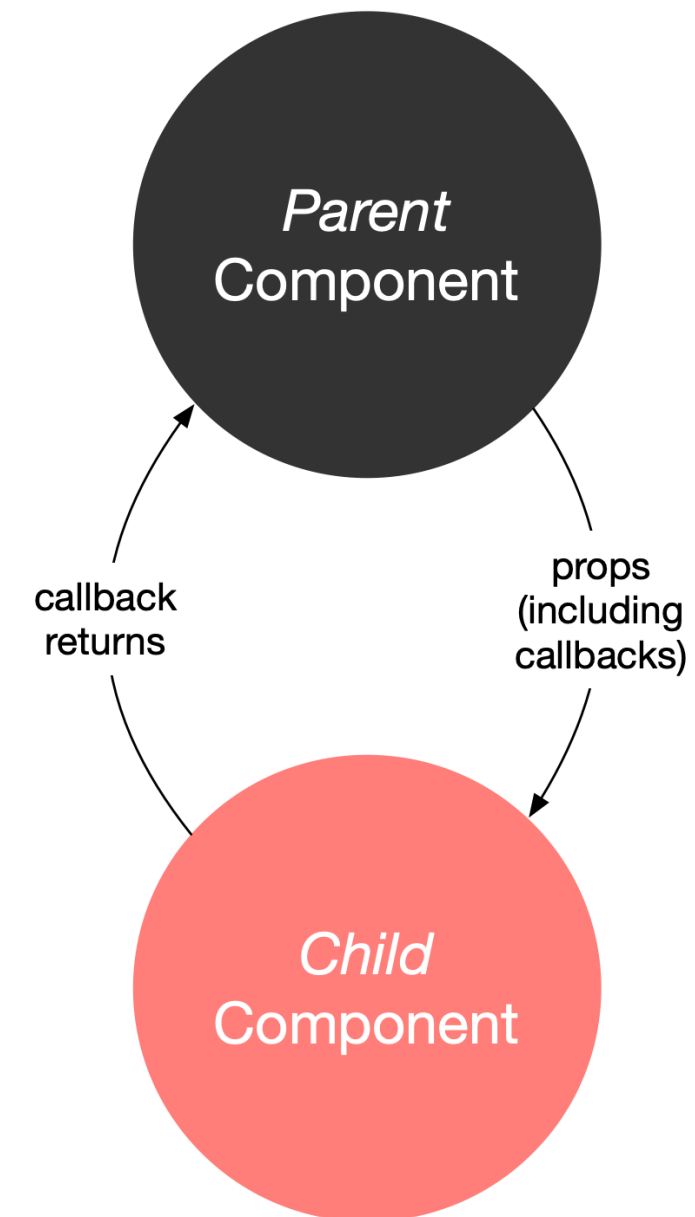
```
<ButtonComponent  
TAContactButtonText={this.state.TAContactButtonText}/>
```



Child-to-Parent Dataflow

This is the more challenging case:

1. Define a callback function in the parent
2. Define a parameter in the callback function for the data that the child will pass
3. Pass the callback as a prop to the child
4. Call the callback using `this.props.<callback-function-name>` in the child and pass the data as the argument



Parent callback:

```
<ButtonComponent callbackFromCard={this.buttonCallback}/>
```

```
buttonCallback = (dataFromButton) => {  
  dataFromButton ?  
  this.setState({ buttonCallbackMessage: "" }) :  
  this.setState({ buttonCallbackMessage: "Connecting..." })  
}
```

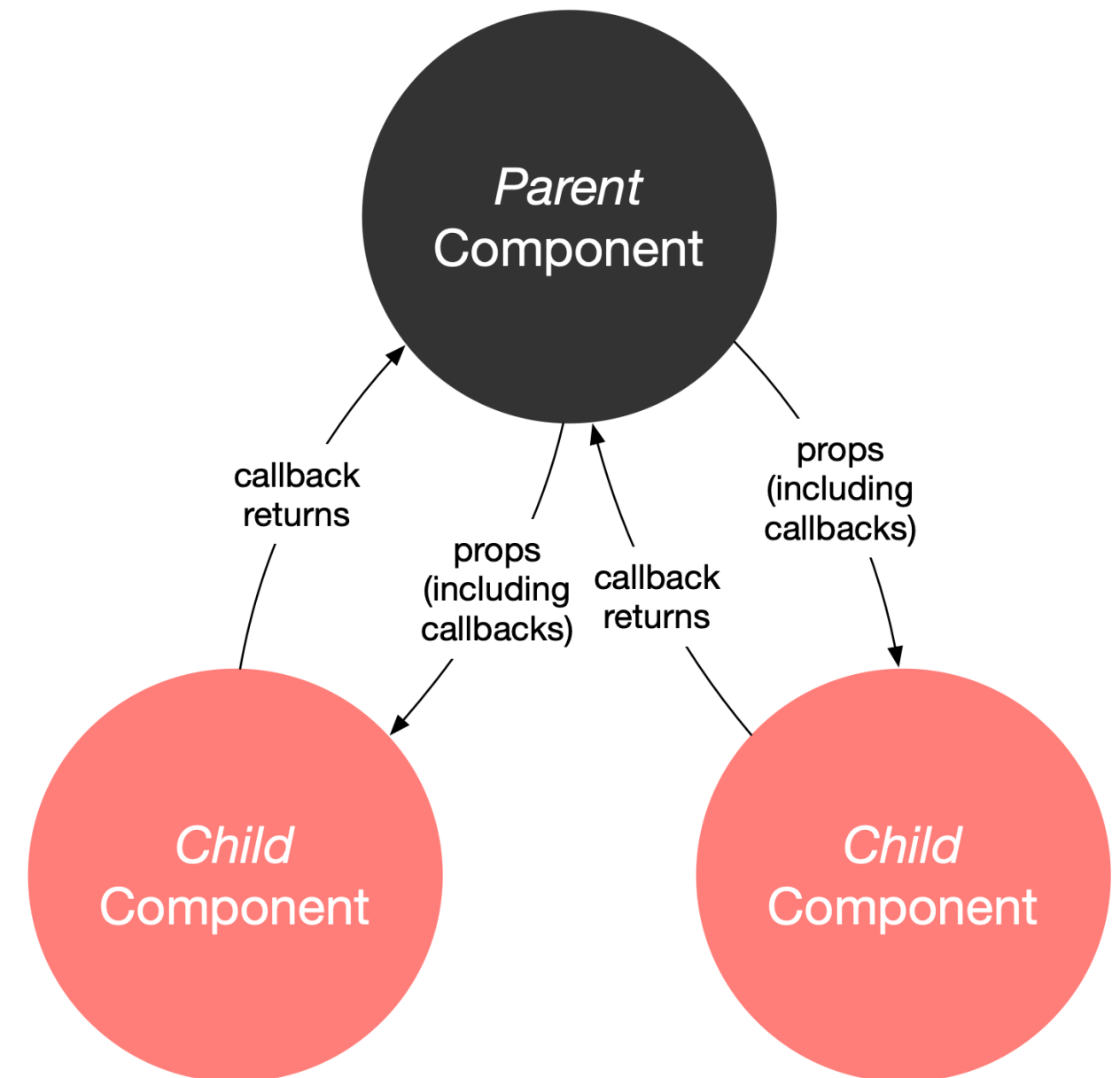
Using the passed callback on the child:

```
this.props.callbackFromCard(this.state.TAContactButtonPressed);
```


Sibling-to-Sibling Dataflow

This case combines both approaches:

1. Identify a parent that can serve as an intermediary
2. Pass data from the child to the parent using a callback
3. Set this data as a state for the parent
4. Pass the data from the state to the other child as prop



Pro Tip: These approaches do not scale well into large applications. Redux (for JS) and React-Redux (for React) help you manage the state of the application. It works like a global object that holds information that can be used across the various components of the application.

Resources:

- [React Redux documentation](#)
- [The only introduction to Redux you'll ever need](#)
- [A complete React Redux tutorial for beginners](#)

What we learned today:

- Using Component Libraries with React
- Component development and reuse
- Dataflow among components